Fast **M**emory **A**llocator

for 32 bits CPU embedded system

FMA32

Table of Contents

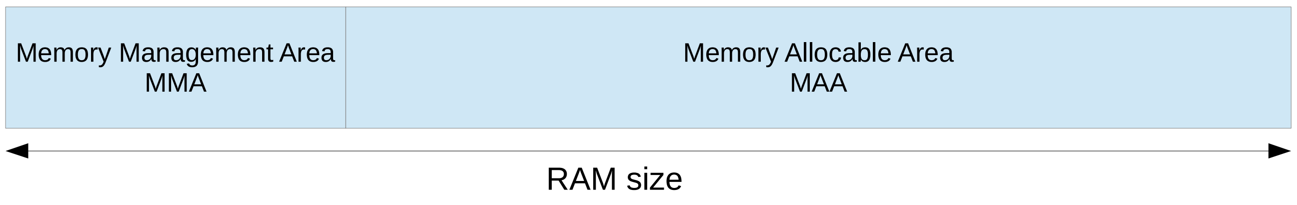
1- Introduction:

A memory allocator is used to reserve memory for specific purposes. Portion of memory allocated is a block with a specific structure. This memory allocator is designed to allocate memory blocks with a very fast constant time. it means allocation and de-allocation take always the same time and don't run through a linked list.

The RAM dedicates to the memory allocator is divided in two main part. The Memory Management Area (MMA) and the Memory Allocable Area (MAA).

The MAA represents the allocable area in the RAM, this is where block of memory are allocated.

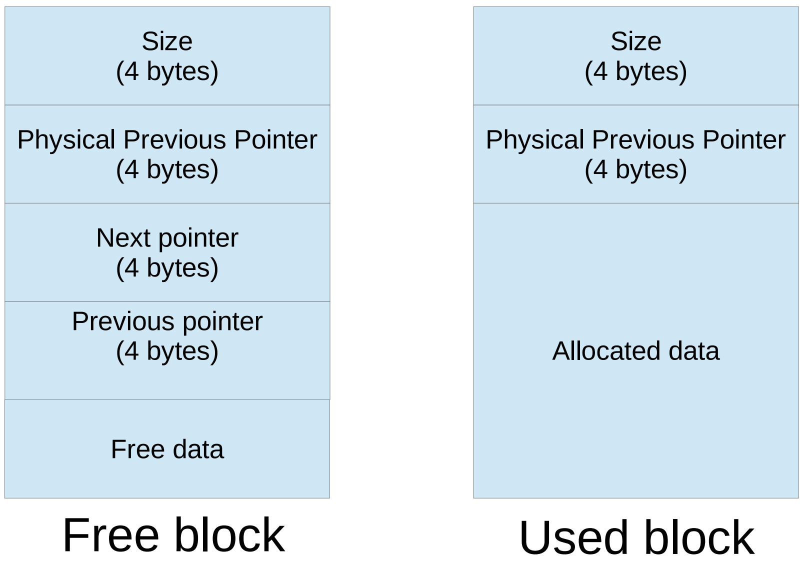
The MMA area contains the system to manage the different lists of free blocks existing in the MAA.



2- MAA : Memory Allocable Area structure

This area contains memory blocks who can be free or used.

2.1- Block used and free structure

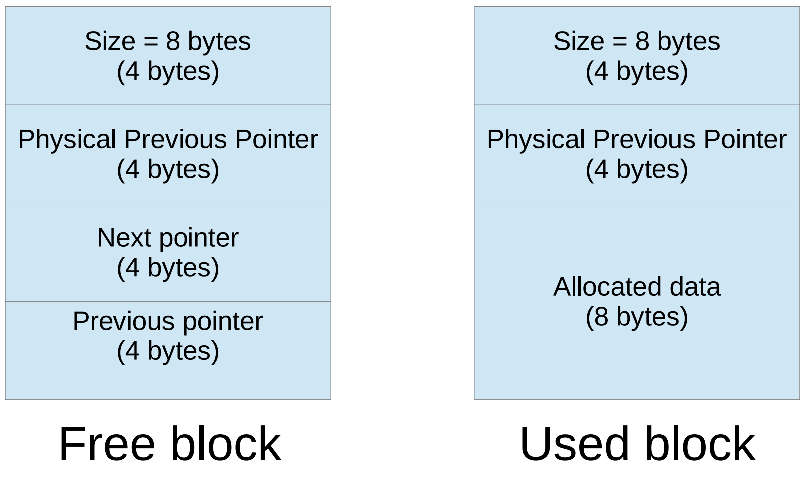


All blocks are aligned on 32 bits size.

2.2- Minimum block size

Minimum block size is fixed by a free block.

This means that an allocated block has a minimum size of 8 data bytes or 16 bytes included the header.



2.3- Block header fields description

2.3.1-Field size

The field “size” indicates the size of the data in the block. This size does not include the 8 bytes of the block header (size field and physical previous pointer).

The last two bits of this field are reserved to indicate respectively if the block is the last and if the block is free or used.

bit 31 … bit 2	bit 1	bit 0
Size in block	0 Not the last block 1 Last block	0 block is used 1 block is free

The “Last Block” bit indicates if the block is the last physical block.

The “Block Free” bit indicates if the block is free or used.

2.3.2-Field Physical previous pointer

The previous physical pointer points to the previous physical block. It is used to know the previous physical block address. There is no needs to have a next physical pointer as the address of the next block can be computed with the current size of the block and his address.

The next physical block address can be calculate like this :

```
Next_block_address = GET_MASKED_SIZE(current_block.size ) + SIZE_OF_HEADER_USED
```

GET_MASKED_SIZE is a macro to get the size without the 2 flag bits.

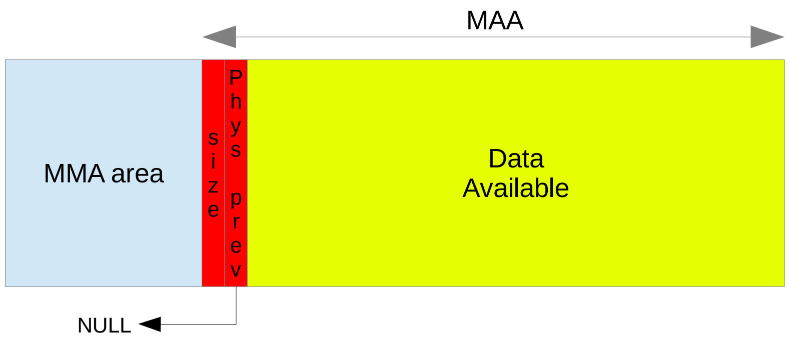
SIZE_OF_HEADER_USED is 8 bytes (4 bytes for the size and 4 bytes for the physical previous pointer).

2.3.3-Field previous and next pointer

Previous and next pointer are used only for free blocks and have no signification for used blocks. These pointers are used to manage the linked lists of free blocks in the MMA area.

2.4- Blocks initialization in the MAA

At the beginning the MAA contains one big free block who covers the whole size.



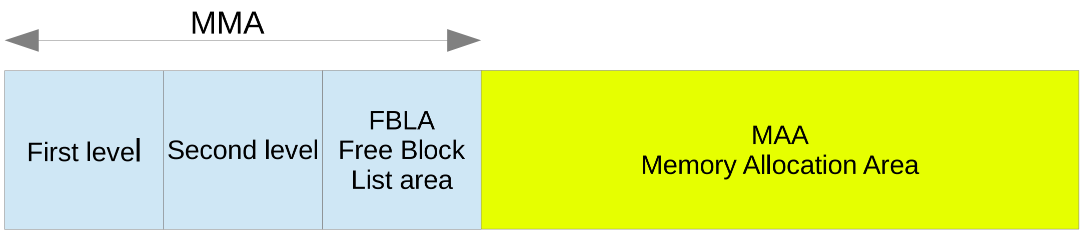
🟨 Data, free space of the block

🟥 Header, block is tagged free, last and physical prev pointer is null

3- MMA : Memory Management Area structure

The MMA is a data structure used to find free blocks in a fastest way in the MAA depending of the required size.

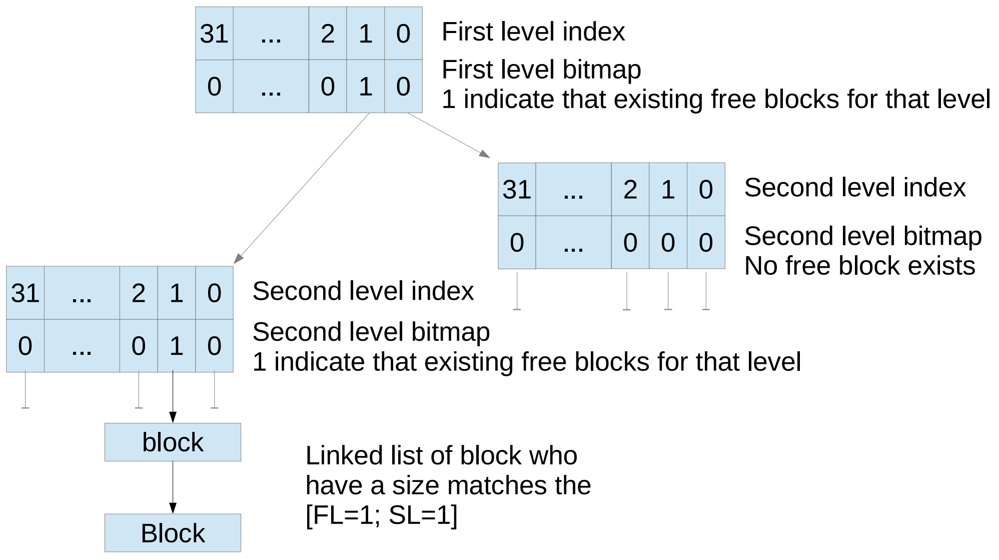
The MMA is divided in 3 parts : the first level (FL), the second level (SL) and the free block lists area (FBLA).



3.1- Principe

To find or register a free block, the MMA manages several linked lists. These lists can be reached using the first and the second level. The first level is a 32 bits bitmap. Each bits represent a range of sizes. Each ranges point to another 32 bits bitmap called second level. This second level divides the first level range size in 32 sub ranges. Each sub ranges contains a pointer to a linked list of free blocks.

When there is no free blocks in the list, the pointer is null.



3.2- The first level

This first level is a 32 bits integer. Each bit represents a range of power of two of sizes (see 3.2.2 Range of size for each bit in the first level).

3.2.1-First level formula

```
If the size is < 128
        First level index = 0;
        First level bitmap = 1;
else
        First level index = bit_highest_pos(size) - 6;
        First level bitmap =  1 << first level index;
```

To avoid any issue with the flags (last and free) the size has to be modulo 4.

3.2.2- Range of size for each bit in the first level

Bit 7 First level 7	Bit 6 First level 6	Bit 5 First level 5	Bit 4 First level 4	Bit 3 First level 3	Bit 2 First level 2	Bit 1 First level 1	Bit 0 First level 0
8KB - 16KB	4KB - 8KB	2KB-4KB	1KB - 2KB	512 - 1KB	256 - 511	128 - 255	0 - 127

Bit 15 First level 15	Bit 14 First level 14	Bit 13 First level 13	Bit 12 First level 12	Bit 11 First level 11	Bit 10 First level 10	Bit 9 First level 9	Bit 8 First level 8
2MB- 4MB	1MB - 2MB	512KB - 1MB	256KB - 512KB	128KB - 256 KB	64KB - 128KB	32KB - 64KB	16KB - 32KB

Bit 23 First level 23	Bit 22 First level 22	Bit 21 First level 21	Bit 20 First level 20	Bit 19 First level 19	Bit 18 First level 18	Bit 17 First level 17	Bit 16 First level 16
512MB - 1G	256MB -512MB	128MB - 256MB	64MB - 128MB	32MB - 64MB	16MB - 32MB	8MB - 16MB	4MB - 8MB

Bit 31 First level 31	Bit 30 First level 30	Bit 29 First level 29	Bit 28 First level 28	Bit 27 First level 27	Bit 26 First level 26	Bit 25 First level 25	Bit 24 First level 24
Not used	Not used	Not used	Not used	Not used	Not used	2G - 4G	1G - 2G

3.3- The second level

A bit of the first level point to a second level. This second level divides the range size of the first level in 32 sub-range of blocks.

Each sub-range is pointing to a linked list of free blocks in its range size.

3.3.1- Second level formula

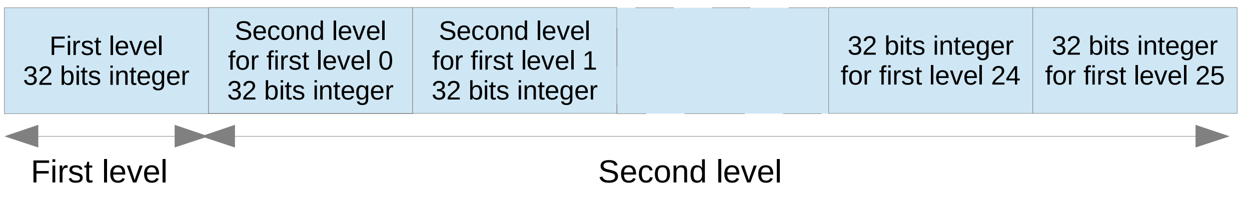
```
If the size is < 128
        Second level = size >> 2;
        Second level bitmap = 1 << Second level;
else
        Second level  = ((size - (fl_bitmap << 6))) / (fl_bitmap >> 1)) >> 2;
        Second level bitmap = 1 << Second level;
```

The variable fl_bitmap is the first level bitmap value.

To avoid any issue with last and free flags, the size has to be modulo 4.

3.3.2-Second level structure

The second level structure is an array of 32 bits integer where each bit represents a sub size range of the first level size range.

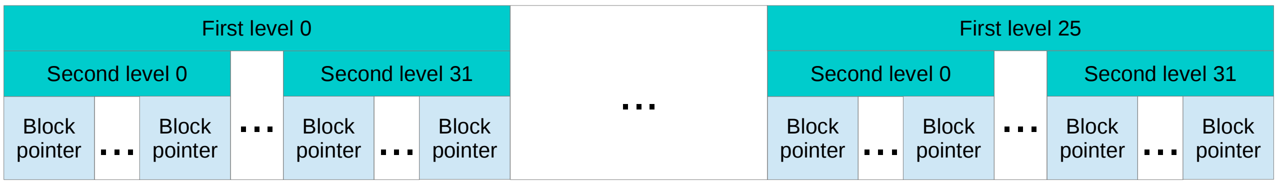


3.4- FBLA : Free block lists area

The free block list area contains all the lists of free available blocks. Bit at one, in the first level bitmap indicates that existing free blocks in this level. The fist level bitmap points to the second level bitmap.

Bit at one in the second level indicates that existing free blocks for that level. The Second level bitmap points to a linked list of free block or null if no free block exists in this level.

The FBLA is an array of array of memory block pointers. FBLA is indexed by the first level and the second level values .



4- Software design

4.1- Public functions

Memory allocator has free public functions :

- void memory_init(void * mem_ptr, unsigned long mem_size)
- void * memory_alloc(unsigned long size)
- void memory_free(void * ptr)

4.1.1-Memory initialization function : memory_init

This function contains two parameters, a pointer to beginning of the RAM and the size of the RAM.

The initialization consists of two operation, first, separates the RAM in two area, the MMA and the MAA and initializes both.

4.1.1.1- Prototype

void memory_init(void * mem_ptr, unsigned long mem_size)

parameter [in] mem_ptr : pointer to the start of the RAM

parameter [in] mem_size : size of the RAM

4.1.1.2- Memory management structure

```
typedef struct memory_management_area_s {
  unsigned long * first_level;
  unsigned long * second_level;
  memory_block_t *(*fbla)[32];
} memory_management_area_t;
```

4.1.1.3- Free block lists area (FBLA):

The size of the MMA is not fixed in order to optimize the size and avoid to lose space. it depends of the size of the RAM given to the initialization function. The address is adjusted by the memory_init function to be aligned of 4 bytes and correct the automatically the size.

The size of the second in byte is given by the formula :

FBLA size in bytes = (bit_highest_pos(length) - 6) * 4;

(length represents the size of the memory given as parameter to the init function)

The size of the FLBA in bytes is given by the formula :

FBLA size in bytes = (bit_highest_pos(length) - 6) * (32 * 4);

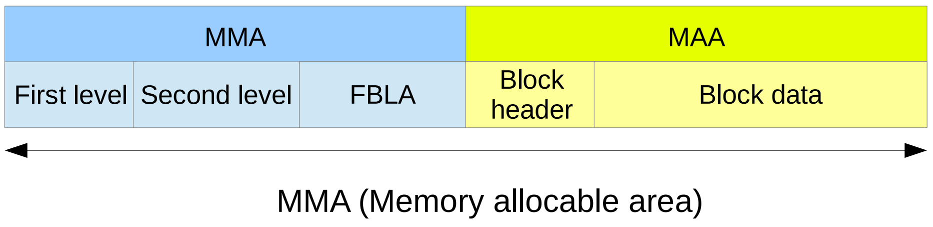
(length represents the size of the memory given as parameter to the init function)

The function bit_highest_pos get the position of the highest bit.

4.1.1.4- Memory allocable area

MAA size is the whole RAM minus the MMA.

The memory_init function initialize the MAA with one block who cover the whole size , see figure below.



The block header contains the size of the block of data with flag free set to 1 and flag last set to 1. The previous physical pointer is set to NULL (no previous block). The function update the MMA to insert this block in the list according is level by an internal function called block_insert (see Block insert function).

4.1.1.5- memory block structure

```
typedef struct memory_block_s {
  unsigned long size;
  struct memory_block_s *phys_prev;
  struct memory_block_s *prev;
  struct memory_block_s *next;
} memory_block_t;
```

See chapter 2.3-Block header fields description

4.1.2-Memory allocation function : memory_alloc

This function is used to allocate a chunk of memory.

4.1.2.1- Prototype

void * memory_alloc(unsigned long size)

Parameter [in] size : size to allocate

This function return the pointer of the chunk of data allocated or null if the memory can't allocate the requested size.

4.1.2.2- Description

The function get the next level according to the size given in parameter. It searches a free block (block_find see 4.2.4) , if found extract (block extract see 4.2.3) the block from the free list and split the block (block_split see 4.2.5).

4.1.3-Memory free function : memory_free

This function is used to free a block previously allocated by memory_alloc.

4.1.3.1- Prototype

void memory_free(void *ptr)

Parameter [in] ptr : pointer previously given by the function memory_alloc

4.2- Private functions

4.2.1-Block get levels function

This function is used to get the first and second level values according to the size. For first and level formulas see chapter 3.2.1 and 3.3.1.

4.2.1.1- Level structure

```
typedef struct {
  unsigned long fl;
  unsigned long fl_bitmap;
  unsigned long sl;
  unsigned long sl_bitmap;
}memory_level_t;
```

The level structure contains values for first and second level. For each there is two type of value the bitmap or the index.

4.2.1.2- prototype

`static void block_get_levels(unsigned long size, memory_level_t *level)`

Parameter [in] size : size used to get the levels.

Parameter [out] level : pointer to the level object to set.

4.2.2-Block insert function

This function update the first level, update the second level and insert the block in the free block list area.

The function marks the inserted block as free. It is existing two cases , first is insert a block in an existing list and second is insert a block in a non existing list.

4.2.2.1- Prototype

```
static void block_insert(memory_block_t * block);
```

Parameter block [in] : pointer to the block to insert in the free list.

4.2.2.2- Insertion in an empty list

If the list is empty, it means that mma.fbla[fl][sl] is null. So, the null pointer is replaced by the block pointer.

4.2.2.3- Insertion in a non empty list

If the list is not empty, it means that mma.fbla[fl][sl] is not null. The block will be inserted at the head of the list for fast insertion.

For both cases, updating levels result in getting the current level of the block size and make a or with first and second level of the MMA.

```
block_get_levels(block->size, &level);
*mma.first_level |= level.fl_bitmap;
mma.second_level[level.fl] |= level.sl_bitmap;
```

Moreover, the block is marked as free block

4.2.3-Block extract function

This function update the first level, update the second level and extract the block from the free block list area. It is existing three cases , first is extract the block when it is the first block of the list, second is when it is the last block and the third it is when the block is between the first and the last block.

4.2.3.1- Prototype

```
static void block_extract(memory_block_t * block);
```

Parameter block [in] : pointer to the block to extract from the free list.

4.2.3.2- Extraction when the block is the first in the list

If the block is not the only element in the list, extract function has to set the next block as the head of the list by update the previous pointer of this next block and updating the FBLA area.

If block is the only element in the list, extract function has to set to NULL the FBLA of the current level to indicate that the list is empty. Moreover it has to upgrade the second level by resetting the bit according to the level. If the second level bitmap is zero, it means that there is no second level for the current first level and resetting the first level bit is mandatory.

4.2.3.3- Extraction when the block is the last of the list

If the block is the last, extract function has to set the next pointer of the previous block to NULL to indicate that it is the last block.

4.2.3.4- Extraction when the block is between the first and the last block of the list

If the block is between the first and the last block of the list, extract function has to update the next pointer of the previous block to the next block and update the previous pointer of the next block to the previous block.

For those three cases, extract function has to set the previous and next pointers of the block to extract to NULL.

4.2.4-Block find

4.2.4.1- Description

This function try to found a free block in the free blocks lists matching the block size.

First it has to get the levels of the size of the block and check if it is existing a free available block. If a block is found, the function has to return the pointer of this block. If there is no existing block at this level the function has to check in the next first upper second level. If no free block is found in the second level, the function has to check if there is a free block in the next first upper first level. If there is no block then it means that there is no possibilty to allocate memory. Then function return null to indicate that it can't find a free block.

4.2.4.2- Prototype

```
static memory_block_t * block_find(memory_level_t *level)
```

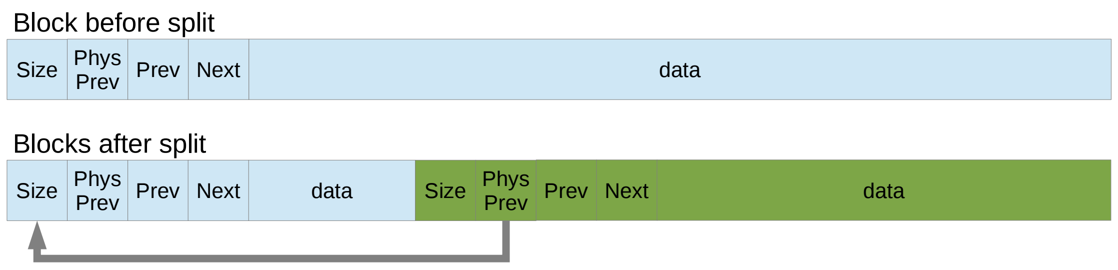
Parameter [in] : level used to find a free block.

Return the pointer of free block found or null if no free block found.

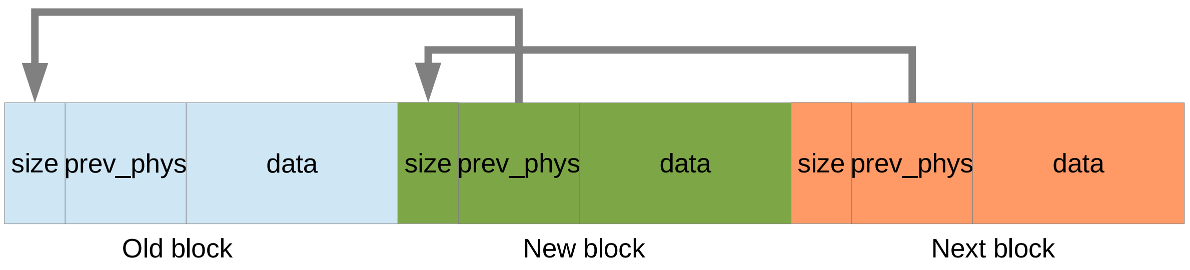
4.2.5-Block split

4.2.5.1- description

This function check if a free block needs to be split during an allocation operation. If the size of the block minus the required size if greater or equal to the minimum block size then the free block can be split and the new block can be inserted in the free block list according to it size.



If the old block is the last physical block (tagged last block) then the new block is tagged as last block and the old block is tagged as not last block. Otherwise if the old block is not the last, the extract function has to update the physical previous pointer of the next block.



4.2.5.2- Prototype

`static void block_split(memory_block_t *block, unsigned long size)`

Parameter [in] block : block to split

Parameter [in] size : size used to split

4.2.6-Block merge right and merge left

4.2.6.1- Description

The both functions are used to merge free blocks during a free operation in order to reduce the memory fragmentation.

4.2.6.2- Block merge left

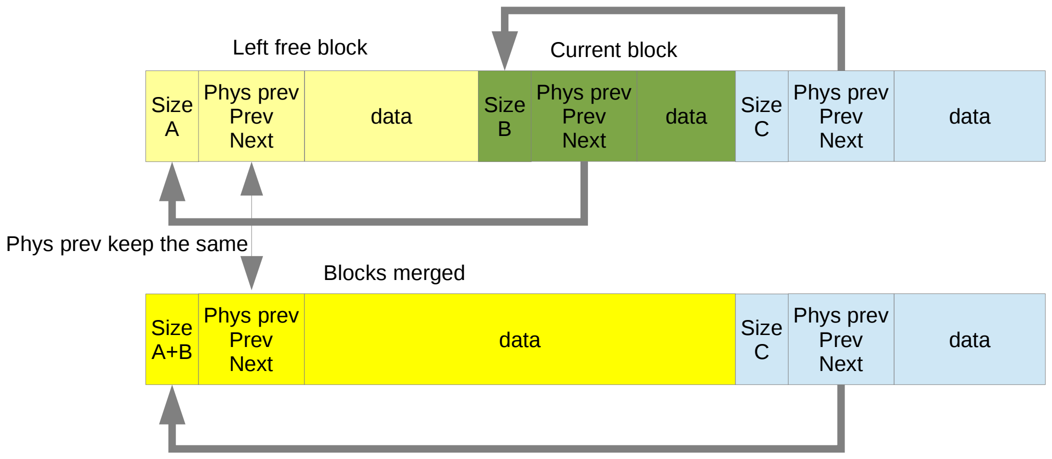
Block merge left execute a merge of the current block and the previous block if this previous block is free.

It existing several cases :

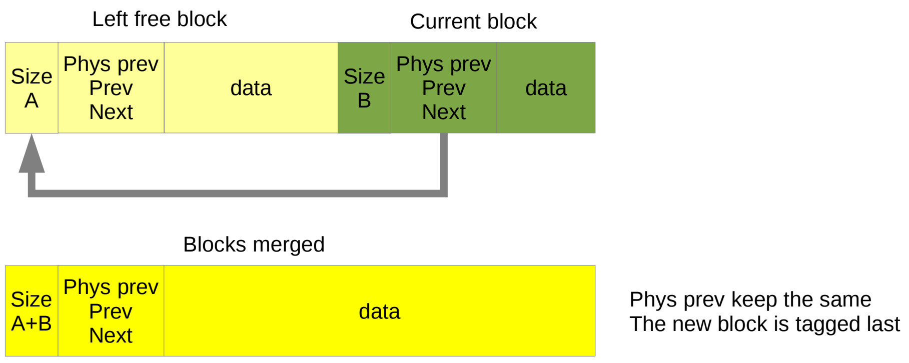
Current block is the first block : no left merge.

Current block is not the first and previous block is not free : no left merge.

Current block is not the first and previous block is free :



Current block is is the last and previous block is free :



4.2.6.3- Block merge right

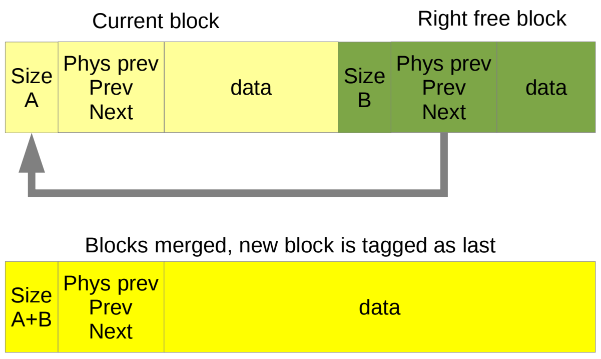
Block merge right execute a merge of the current block and the next block if this next block is free.

It existing several cases :

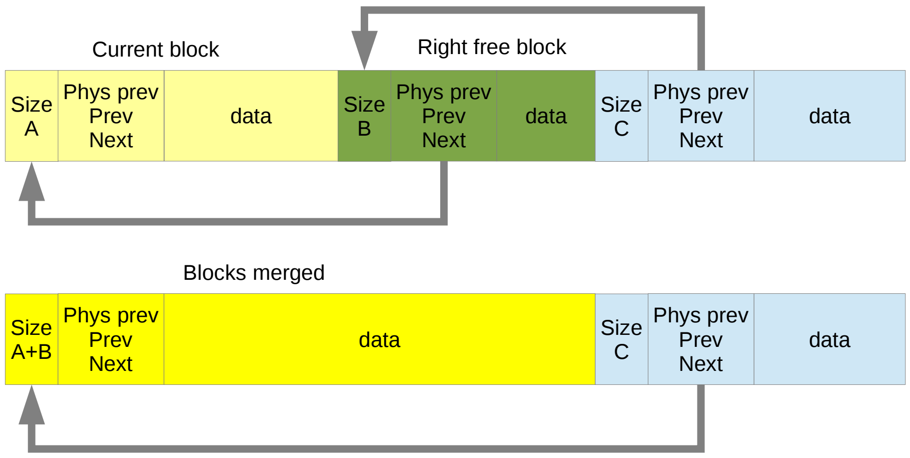
Current block is the last block : no right merge.

Current block is not the last and next block is not free : no right merge.

Current block is not the last, next block is free and it is the last :



Current block is not the last, next block is free and it is not the last :


Current block
Right free block
Size A
Phys prev Prev Next
data
Size B
Phys prev Prev Next
data
Size C
Phys prev Prev Next
data
Blocks merged
Size A+B
Phys prev Prev Next
data
Size C
Phys prev Prev Next
data